Latency-Insensitive Design and Central Repetitive Scheduling

Julien Boucaron, Jean-Vivien Millo, Robert de Simone

▶ **To cite this version:**

Julien Boucaron, Jean-Vivien Millo, Robert de Simone. Latency-Insensitive Design and Central Repetitive Scheduling. [Research Report] RR-5894, INRIA. 2006. inria-00071374

HAL Id: inria-00071374
https://inria.hal.science/inria-00071374

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Latency-Insensitive Design and Central Repetitive Scheduling

Julien Boucaron — Jean-Vivien Millo — Robert de Simone

N° 5894

Avril 2006

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5894--FR+ENG

Latency-Insensitive Design and Central Repetitive Scheduling *

Julien Boucaron , Jean-Vivien Millo , Robert de Simone

Thème COM —Systèmes communicants
Projet Aoste

Rapport de recherche n° 5894 —Avril 2006 — 14 pages

Abstract: The theory of latency-insensitive design (LID) was recently invented to cope with the *time closure* problem in otherwise synchronous circuits and programs. The idea is to allow the inception of arbitrarily fixed (integer) latencies for data/signals traveling along wires or communication media. Then mechanisms such as shell wrappers and relay-stations are introduced to "implement" the necessary back-pressure congestion control, so that data with shorter travel duration can safely await others when they are to be consumed simultaneously by the same computing element. These mechanisms can themselves be efficiently represented as synchronous components in this global, asynchronously-spirited environment.

Despite their efficient form, relay-stations and back-pressure mechanisms add complexity to a system whose behaviour is ultimately very repetitive. Indeed, the "slowest" data loops regulate the traffic and organize the traffic to their pace. This specific repetitive scheduling has been extensively studied in the past under the name of "Central Repetitive Problem", and results were established proving that so-called k-periodic optimal solutions could be achieved. But the "implementation" using typical synchronous circuit elements in the LID context was never worked out.

We deal with these issues here, using explicit representation of schedules as periodic words on $\{0,1\}^\star$ borrowed from the recently theory of *N-synchronous* systems.

Key-words: Latency Insensitive, Relay Station, Shell, GALS, Formal Verification, Marked Graph, Synchronous, Esterel, SyncCharts, Scheduling

* Work partially supported by ST Microelectronics and Texas Instruments grants in the context of the french regional PACA CIM initiative

Le problème central répétitif et les systèmes insensibles à la latence.

Résumé : La théorie des systèmes insensibles à la latence a été récemment introduite afin de traiter le problème dit de *time closure* dans les circuit et programmes synchrones. L'idée est d'autoriser l'insertion de latences arbitraires fixes (entières) pour les données/signaux se propageant le long des fils ou média de communication. Des mécanismes spécifiques dénommés *Shells* et *Relay-Stations* sont introduits afin d'implanter le protocole de gestion de la congestion, de telle manière à ce que les données avec des temps de transport plus rapides puissent patienter et être stockées en attendant les autres valeurs n'ecessairesaux calculs. Ces mécanismes peuvent eux-mêmes être représentés efficacement par des composants synchrones dans cet environnement globalement asynchrone.

Malgré leur efficacité, les *Relay-Stations* et les mécanismes utilisés lors du contrôle de flot ajoutent une complexité non-négligeable au circuit, dont le comportement est par ailleurs ultimement répétitif. Ces ordonnancement répétitifs ont été étudiés en détail dans le passé sous le nom de *Problème Central Répétitif*, et des résultats ont été établis démontrant que des solutions optimales *k-périodiques* peuvent être obtenues. Par contre l'implantation utilisant des circuits synchrones dans les contexte des systèmes insensibles à la latence n'ont pas été effectués.

Nous traitons ces problèmes ici, en utilisant une représentation explicite des ordonnancements avec des mots périodiques sur $\{0,1\}^\star$ empruntés à la théorie récente des systèmes *N-Synchrones*.

Mots-clés : Latency Insensitive, Relay Station, Shell, GALS, Vérification Formel le, Synchrone, Esterel, SyncChart, Ordonnancement

1 Introduction

Today's SoC design faces the problem of *timing closure* and *clock synchronization*. Latency Insensitive Theory [14, 13, 12, 4] was introduced to tackle the timing closure issue. It does so by defining specific storage elements to install along the wires, which then divide signal and data transport according to necessary latencies. Dynamic scheduling schemes are implemented by additional signals which authorize behaviors whenever data have completed their journey.

On the other hand, the general theory of weighted marked graph teaches us that there exist static repetitive scheduling for such computational behaviors [6, 1]. Such static k-periodic schedulings have been applied to software pipelining problems [10, 15], and later to SoC LID design in [5]. But these solutions pay in general little attention to the form of buffering elements that are holding values in the scheduled system, and their adequacy for hardware circuit representation. For instance in [5] the basic clock has to be divided further into multiphases, a solution which we find undesirable here. Also, in these solutions the precise activity allocation of clock cycles to computation nodes is not explicit (except in the restricted case of nested loop graphs).

Expressing an explicit precise static scheduling that uses predictable synchronous elements is desirable for a number of issues. It could easily be synthesized of course, but could also be evaluated for power consumption, by attempting to interleave as much as possible the computation phases with the transportation phases. It could also be used as a basis for the introduction of control modes with alternative choices that are currently absent of the model (it performs over and over the same computations in a partial order). We shall not consider these issue here, but they are looming over our approach to modeling the schedules.

Our contributions Our main objective is to provide a statically scheduled version of relaxed synchronous computation networks with mandatory latencies. It can be thought of as trying to add more latencies to the already imposed ones, in order to slow down tokens to exactly regulate the traffic. The new latencies are somehow virtual, as they are not uniquely defined and could be swallowed by more relaxed *redesign* of computation elements. The goal is to equalize all travel times (counted in clock cycles) in a relaxed synchronous firing rule, so that all flow-control back-pressure mechanisms can be done without, and relay-stations are simplified to mere single-slot registers. But sadly this is not always feasible, due to the fact that exact solutions would require sometime rational (rather than integer) delays to be inserted.

The solution should in then be constructed by inserting specific modeling elements that can be represented by hardware components, in the same way as relay-stations and shell wrappers. Preliminary attempts were conducted in [5], but they need to divide clock cycles into smaller phase, and we do not want this. We need to introduce elements that "capture" tokens to hold them less than each time. The pattern of hold-ups thus has to be made explicit.

In order to represent the explicit scheduling of computation node activities we borrow from the theory of N*-synchronous* processes [7], where such notions were introduced. We identify a number of interest in relative phenomena occurring when loops with rates that do not match (for instance involving prime numbers) are present. We shall also face the issues of the initialization phase, and of the recognition of the stationary cases.

Paper structure: In the next section we recall the *modeling framework* of computation nets and its semantics variations over the firing rules. We start with a recap of the now classical fully synchronous and fully asynchronous semantics. The introduction of weights (or *latencies* brings an intermediate model that lags in-between (retimed asynchronous or relaxed synchronous), with some slight distinctions remaining between them. We provide syntax to express explicit schedules, borrowed from the new theory of N-synchronous processes. It helps us phrase the problem of general equalization of latencies that we want to tackle. We end the section with a summary of important known results on k-periodic static schedules for Weighted Event graphs.

The following section describes our approach and the sequence of algorithmic steps of analysis required to introduce integer latencies, compute the schedules over the initialization and the stationary periodic phases of the various computation nodes, and identify locations where to add extra fractional latencies to even out fully the various data routes between computations. The goal is to maintain the throughput of the slowest loop cycle, which is the best attainable anyway. A formal synchronous description of the *Fractional Registers* involved is provided, with conditions on their application. Matters in efficient implementation are also mentioned (while we are currently building a tool prototype around these implementation ideas, it was not fully operational by submission time).

We provide a number of examples to highlights the current difficulties, and we end up with considerations on (numerous) further topics.

2 The Modeling Framework

2.1 Computation nets

We shall loosely call *computation net scheme* a graph structure consisting of computation nodes, linked together by directed arcs. In such a simple model computations should consist in repeatedly consuming input values from incoming arcs, and producing output values on outgoing ones. There are also primary input and output arcs (without a source or target node respectively), and possibly loops in the graph. Behaviors do not in fact depend on actual data values, but only on their availability to computing nodes. They can be abstracted as presence token. The number of tokens in a loop stays invariant across computations.

The model is incomplete without a description of the firing rule, which enforces the precise semantics for triggering computations. The way token may (or may not) be stored at arcs between computations is an important part of these potential semantics. (Partial) computation orderings should enforce obvious properties: informally stated, production and consumption of data should ultimately match in rates. No data should be lost or overwritten by others, deadlocks (data missing) and congestions (data accumulated in excess) should be avoided altogether. Figure 1(a) displays a simple computation net scheme.

2.2 Synchronous, asynchronous and relaxed-synchronous semantics

There are two obvious starting points for possible semantics: the *fully synchronous* approach, in which all computations are performed at each cycle of a global clock,so that data are uniformly

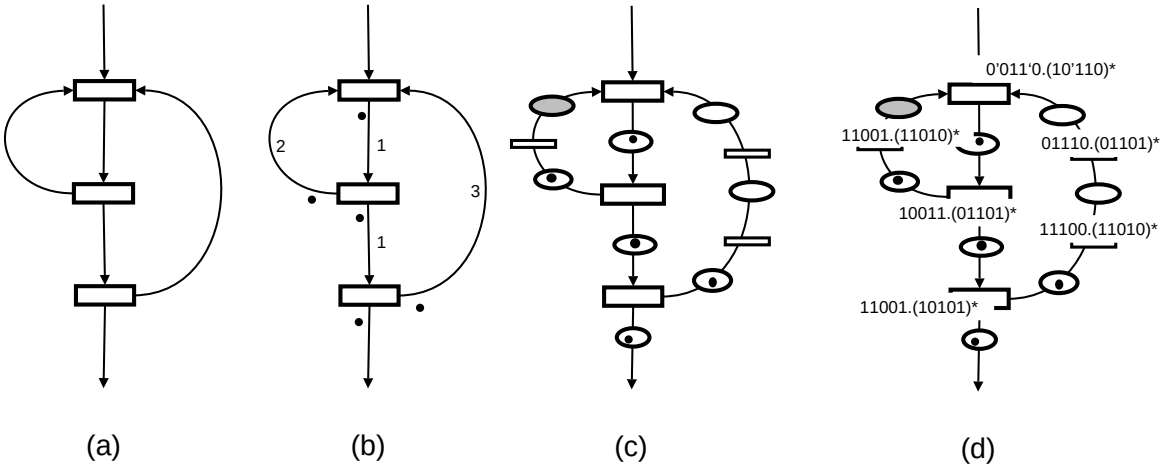

Figure 1: (a) An example of computation net, (b) with token marking and latency features, (c) with relay-stations dividing arcs according to latencies, (d) with explicit schedules

flowing; the *fully asynchronous* approach, with infinite buffers to store as many data as needed in between independent computations. The first approach is represented in the theory of synchronous circuits and synchronous reactive languages [2], the second in the theory of *Event/Marked graphs*[8]. Both lead to a profusion of theoretical developments in the past. In both case there is a simple correctness assumption to guarantee safety and liveness: the existence of (at least) a unit delay element in each loop in the synchronous case, the existence of (at least) a token in each loop in the asynchronous case. Recall that the number of data cannot blow up inside loops in the asynchronous case, as it remains invariant.

Fully synchronous and *fully asynchronous* semantics are somehow extreme endpoints on the semantic constraint scale. As a middle point proposals have been made to introduce explicit *latencies* on arcs to represent mandatory travel. In such a simple model what matters most is the potential time for data/tokens. Starting from the asynchronous side, this lead to the theory of *Weighted Event/Marked Graphs* (WEM graphs), extensively studied in the past [1, 6] to model production systems. Starting from the synchronous side, this led to the theory of *Latency-Insensitive Design* (LID) [4], which attempts to solve the so-called *Timing Closure* issue in circuits where electric wires can no longer be assumed to propagate in unit time.

Figure 1(b) displays a computation network annotated with latencies. Figure 1(c) refines this description by making explicit the successive stages in the travel and the places where tokens may reside while on travel (the smaller rectangles can be thought of as *motion operations*, similar to the computation as the consume and produce token from one place to another). For the sake of simplicity we shall assume for now that all places following a computation nodes are marked by a token (and

only them), as shown on figure 1(c). Weaker assumptions can be taken, reflecting only the needs expressed previously (at least one token on every loop cycle).

Studies on both LID networks and WEM graphs both attempt at providing synchronous execution rules to the computation nets, in the sense that all computation nodes fire as soon as possible, possibly simultaneously. Their setting and motivations still differ in several ways:

- In LID theory [13, 12] the buffering places are replaced by so-called *relay-stations*, and the computation nodes are surrounded by *shell wrappers*. The purpose of these components, which are described as additional *synchronous* elements [3], is to implement a dynamic online scheduling scheme: it regulates data traffic to the point that never more than two tokens accumulate in any relay-station. Computations require all input tokens, but also free space in output relay-stations. In the example of figure 1 (c), in the second step of th einitialization phase the relay-station in the grey place would need to hold two token values (while the token on the right arc travels up).
- Research conducted on WEM graphs attempts to obtain static repetitive scheduling, based on the fact that a synchronous firing of computation nodes leads to a deterministic behavior, which is bound to repeat itself with a given period because of the finiteness of possible token distributions. But the scheduling does not pay much attention to the distribution of token in between places (and thus the buffer sizes). The schedules are ultimately k-periodic of period p, meaning that a transitory phase (initialization) of finite length is followed by a periodic stationary phase, which consists for each computation node of a repetitive pattern of k active instants over a period of p instants.

In both cases there is no guarantee that the actual computation rate will match the one indicated by the user-provided latencies. Data with shorter travel time may have to wait for others on other routes to allow computations. In that sense the latencies indicated form a floor value to the effective ones. Both models identify an important notion of *rate* to speak of the computational frequency. Classical results on WEM graph scheduling are summed up in section 2.4.

Definition 1 (Rates and critical cycles). Let G be a Weighted Marked graph, and C a cycle in this graph.
The *rate* r of the cycle is equal to $\frac{T}{L}$, where T is the number of token in the loop (which is constant), and L is the sum of latencies labeling its arcs.
The *throughput* of the graph is defined as the min of rates over all cycles.
A cycle is called *critical* if it srate is equal (i.e, as slow) as the graph throughput.

2.3 Explicit schedules

We borrow from the theory of N-synchronous processes [7] a notation to represent explicitly the types of schedules that are aimed at, namely the k-periodic ones.

Definition 2 (Schedules). A *schedule* w is a word $u.v^\star$ where $u, v \in \{0,1\}^\star$. u represents the initial part, v the periodic one. We note $length(v) = p$.
The schedule *rate* r_w can be defined as $\frac{|v_1|}{|v|} = \frac{k}{p}$, where $|v_1|$ is the number of occurrences of 1s in v.

The intuition is that a schedule forces activity at instants where it holds a "1", and inactivity when "0". In the theory of N-synchronous processes such schedules are used to harmonize parallel process with schedules with identical rates, but possibly shifted locally at places (see again section 2.4 for a summary). We shall use them to decorate computation nodes. It allows us to define our goals in a formal way.

Figure 1(d) displays the schedules obtained by running the former example. Ideally, if the system was well-balanced on latencies, the schedule of a given computation nodes should exactly be the one of its predecessor shifted right by one position (in a modulo fashion). But when data do *not* arrive synchronously at a given computation nodes and some have to be stalled at the entry, it is not the case. In our example this can occur only at the topmost computation nodes. Here we prefixed some of the inactivity 0 marks by a symbol to indicate that inaction is due to a lack of input token from the right (’), or on the left (‘).

The way we seek to solve the problem is by adding new latencies to some of the arcs (the fastest), to slow them down to the rate of the slowest. But while this can be solved by encoding and linear programming techniques, the exact solution is not always of *integer* nature. For instance on our example, the leftmost loop is of rate $2/3$, and the (slower) rightmost one is of rate $3/5$. But adding one extra integer latency to the left loop brings its rate down to $\frac{2}{3+1} = 1/2$, strictly slower than the former slowest one, and thus reducing the global achievable throughput. It has been proposed recently to cope with the issue by allowing k-*phase* clocks, whereby the unit time is divided into smaller granular instant by oversampling. We want to avoid this, which is hard to put into practice. Instead we try to add as many full integer latencies as possible, and add only fractional delay elements, that hold back less than each token one unit of time. These elements should be placed in specific locations: in our example the place painted grey is a natural candidate. They should be described as synchronous components, following the trend of relay-stations and shell wrappers. They should correspond to schedules (for holding back tokens sometimes, sometimes not), so that the global informative scheduling scheme is now well-balanced. This means that tokens exactly arrive simultaneously at a node to be computed upon.

2.4 Related works and known results

The foundations of the theory of static and k-periodic scheduling for Weighted Marked Graphs is to be found in [6, 1]. In [6] the authors name it as the *Central Repetitive Problem (CRP)*. It is formulated as a generic scheduling problem :given a set of tasks, and constraints between them and between different runs of the same task, the goal is to execute this set of tasks infinitely while minimizing time spent for each execution.

In [1] it is established that the scheduling of a connected graph with cycles G, is ultimately k-periodic. A bound for k is established as equal to the cyclicity of G^c, where G^c represents the restriction of G to its critical cycles). The cyclicity of G^c is equals to the lcm (Least Common Multiple) of the cyclicity of each SCC (Strongly Connected Component) of G^c. The cyclicity of a SCC of G^c is equals to the gcd (Greatest Common Divisor) of the token count of all its cycles.

In contrast to these results on the stationary periodic phase, little is known on the length of th einitialisation phase that leads to it.

Example : the figure 2 is a graph G. It has three cycles and only $c1$ and $c3$ are critical. The cyclicity of the first SCC of G^c, formed with $c1$ is 1. The cyclicity of the second SCC of G^c, formed with $c3$ is 2. The Cyclicity of G^c is $lcm(1,2) = 2$. The graph is 2-periodic with a period 4. Note than $c1$ is 1-periodic with period 2. It exists two proportional sizes of k-periodicity because critical cycles are disjoints.

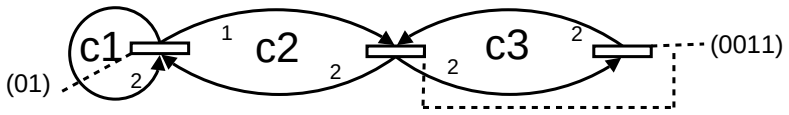

Figure 2: Example of a graph, calculation of the periodicity and the schedule of each node

In [10] the authors introduce a time optimal algorithm assuming unbounded resources in the case of software pipelining. In [15] the authors use this result in the context of synchronous circuits; but they rely on the use a multi-phase clock division for the *rational solutions* issue. In [5] the authors applies a variation of the previous algorithms to LID systems; they also obtain a static schedule with the previous properties, but again with the help of multi-phase clocks, that are usually not the way meant to implementation.

3 Equalization Process

In our attempt to design a static predictable scheduling for the whole systems, we are now describing the successive algorithmic steps involved in the process of equalizing the various loop latencies. These steps consist in: *determining the expected global throughput*; *computing a maximal set of integer latencies* (to add corresponding elements to the system description; *computing the transitory and stationary schedules* (by state space construction); adding the fractional latency elements. Other (optional) algorithmic techniques can be studied to shorten the initialization sequence of computation and transportation steps, making it asynchronous. We further detail these steps below.

Global throughput computation. We are here seeking to compute the best admissible rate, that is the rate of the slowest cycle (noted R). This is done by enumerating cycles, and is related to the so-called *Minimal Cost-to-Time Ratio Cycle problem in graph theory and operational research [11, 9].*

Integer latency insertion. This is solved by linear programming techniques. Extra latency variables are added to the existing latencies of the non-critical cycles. Linear equation systems are formed to indicate that all elementary cycles should now have the global throughput as their rate (this is a reason why all elementary cycles had to be explicitly enumerated before). The solution gives rational values for results as additional latencies. Such a value x can of course be partitioned into its integer part $q = \lfloor x \rfloor$ and a fractional part $f = x - q$, with $0 \leq f < 1$. We keep the qs as

additional latencies. An additional requirement to the solver can be that the sum of added latencies be minimal (so they are inserted in a best factored fashion).

Alternatively one could compute for each cycle, given its number of token T, its longest integer latency achievable without slowing down the whole system, which is $\lfloor \frac{T}{R} \rfloor$. But the counter-example 3 shows a difficulty here.

State space construction. Then to compute the schedules of the initial and stationary phases we currently need to *run* the system and build the state space. The purpose is to build (simultaneously or in a second phase) the schedule patterns of computation nodes, including the quote marks (') and (‘), so as to determine where residual fractional latency elements have to be inserted in the next phase.

In a synchronous run each state will have only one successor, and this process stops as soon as a state already encountered is reached back. The main issue here consists in the state space representation (and its complexity). In a naive approach each place may hold T tokens, where T is the minimal sum of tokens over all elementary cycles that use this place. But with the extra latencies now added, we can use LID modeling, with relay-stations and back-pressure mechanisms. Then each place can hold at most two tokens, encoded by two boolean values. Then a global state is encoded as $2n$ boolean variables, where n is the sum all all latencies over the arcs. Further simplification of the state space in symbolic BDD model-checking fashion is also possible due to the fact that, in between the two values describing a relay-station state, one can be filled only if the other one already is. This is explained in further details below.

We are currently investigated (as “future work”) analytic techniques so as to estimate these phases without relying on this state space construction.

Fractional latencies. We then need to introduce a synchronous element to equalize the remaining fractional differences in rates wherever reconvergence of flows slightly out-of-phase occurs. Such a element could be made to hold only one token at a time, and to hold them (or not) according to a pattern that can be represented as a schedule of the same nature as the ones of computation nodes. We conjecture that, after integer latency equalization, such elements are only required just before computation nodes (that is, we do not need several in sequence as relay-stations do). We discuss this issue further below.

Optimized initialization. So far we have only considered the case where all components did fire as soon as they could. Sometimes delaying some computations or transportations in the initial phase could lead faster to the stationary phase, or even to a distinct stationary phase that may behave more smoothly as to its scheduling. Consider in the example of figure 1 (c) the possibility of firing the lower-right transport node alone (the one on the backward up arc) in a first step. By this one reaches immediately the stationary phase (in its last stage of iteration).

Initialization phases may require a lot of buffering resources temporarily, that will not be used anymore in the stationary phase. Providing short and buffer-efficient initialization sequences is thus a challenge. We are currently experimenting with symbolic *asynchronous* simulation (something akin to model-checking), trying to reach a given state known to be a stationary source as soon as possible. The asynchronous firing rule allows to perform various computations independently, so

that the tokens may progress to better locations. Then a careful study of various paths may help choose the ones that use the less buffering resources. Other perspectives are open for future work here.

When applying these successive transformation and analysis steps, which may look quite complex, it is often the case in practice that simple sub-cases arise, due to the well-chosen numbers provided by the designer. Exact integer equalization is such a case. The case when fractional adjustments only occur at reconvergence to a critical paths are also noticeable.

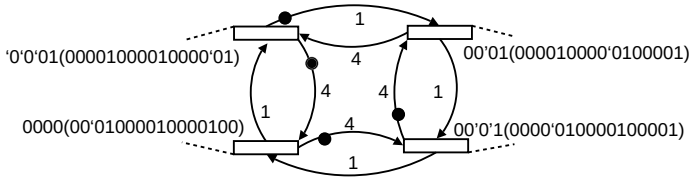

Figure 3: An example where no integer latency insertion can bring all the cycle rates very close to the global throughput. While the global throughput is of $\frac{3}{16}$, given by the inner cycle, no integer latency can be added to the outside cycle to bring its rate to $\frac{1}{5}$ from $\frac{1}{4}$. Instead four fractional latencies should be added (in each arc of weight 1).

3.1 Fractional Register element (FR)

We describe here the specific synchronous elements used to hold back specific tokens for one instant, so that rates are equalized and the tokens are presented simultaneously to the computation nodes. The desired effect is to compensate for the partial stalls recorded by the (') and (') marks (in the binary case).

The FR interface (see figure 4 b) consists of two input wires *TokenIn* and *Hold*, and one output wire *TokenOut*. Its state consists of a (one-slot) register *CatchReg*, used to "kidnap" the token (and its valuein a real setting) for one clock cycle when *Hold* holds. We note $pre(CatchReg)$ the (boolean) value of the register computed at the previous clock cycle. It indicates whether the slot is occupied or free.

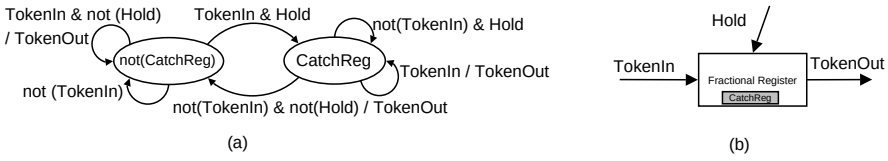

Figure 4: The automaton (a) and the interface block-diagram of the FR element

It is possible that the same token is held several instants in a row (but meanwhile there should be no new token arriving at the element). It is also possible that a full sequence of consecutitive tokens are held back one instant each in a burst fashion (but then each token should leave the element to the computation node while the new one enters). In other words when $Hold \wedge pre(CatchReg)$ holds, then either $TokenIn$ holds, in which case the new data token enters and the current one leaves (by scheduling consistency the computation nodes that consumes it should then be active), or $TokenIn$ does not hold, in which case the current token remains (and, again by scheduling consistency, then the computing node should be inactive). Furthermore the two extra conditions are requested:

$Hold \Rightarrow (TokenIn \vee pre(CatchReg))$**:** if nothing can be held, the scheduling does not attempt to;

$(TokenIn \wedge pre(CatchReg)) \Rightarrow Hold$**:** otherwise the two tokens could cross the element and be output simultaneously.

The FR behavior is provided by the two equations (from the FSM in figure 4 a):

$CatchReg = Hold$**:** the register slot is used only when the scheduling demands;

$TokenOut = (TokenIn \wedge \neg Hold) \vee \big((TokenIn \wedge pre(CatchReg) \wedge Hold\big) \vee (\neg TokenIn \wedge pre(CatchReg) \wedge \neg Hold)$**:** either a new value directly falls across, or an old one is chased by a new one being held in its place.

In many ways our FR model looks like a modeling of (only) the additional auxiliary stage present in a relay-station. Thus by adding (at most) one FR element to simle registers relaizing latency steps one gets the same buffering capacity as with relay-stations; the main difference is that the schedules are now explicit, and provided from outside in accordance to the global flow. Its $Hold$ input can be given a static schedule by our process. It can also be expressed at the entry point of a computation node as a combinatorial formula indicating the need to hold the value as expressed by the other flows arriving at this computation node.

3.2 Correctness issues

As already mentioned we still do not have a proof that in the stationary phase it is enough to include such elements at the entry points of computation nodes only, so they can be installed in place of more relay-stations also. Furthermore it is easy to find initialization phases where tokens in excess will accumulate at any locations, before the rate of the slowest loop cycles distribute them in a smoother, evenly distributed pattern. Still we have several hints that partially deal with the issue. It should be remembered here that, even without the result, we can equalize latencies (it just needs adding more FR elements).

Definition 3 (Smoothness). A schedule is called smooth if the sequences of successive 0 (inactive) instants in the schedule in between two consecutive 1 cannot differ by more than 1. The schedule $(1001)^\star$ is *not* smooth since they are two consecutive 0 between the first and second occurrences of 1, while there is none between the second and the third.

Property 1. *If all computation node schedules are smooth, rates can be equalized using FR elements only at computation node entry points.*

Proof. Suppose a vertex have token on standby in FR of input(s), but some token on other input(s) are absent(s). So, the maximal waiting time of absent token (n) is inferior or equal to the minimal distance (in clock cycle) between a present token and its following (n). In the worst case, when the following arrive, all previous are present and fire the transition. The condition of correctness on FR is preserved, a storage capacity of 1 is enough. □

4 Further Topics

We are currently implementing a prototype version. While some of the graph-theoretic algorithms are well-documented in the literature, the phase of symbolic simulation and state-space traversal needs careful attention to the representation of states (as Binary Decision Diagrams on variables encoding the local Relay-station states). Also the representation of schedules must be tuned. They are really strictly needed only at computation nodes, and can be recovered on demand at other places (the transportation nodes).

A number of important topics are left open for further theoretical developments:

- We certainly would like to establish that FR elements are needed only at computation nodes, minimizing their number rather intuitively;
- Discovering short and efficient initial phases is also an important issue here;
- The distribution of integer latencies over the arcs could attempt to minimize (on average) the number of computation nodes that are active altogether. In other words transportation latencies should be balanced so that computations alternate in time whenever possible. The goal is here to avoid *"hot spots"*. It could be achieved by some sort of retiming/recycling techniques;
- While we deal with *transportation* latencies, in general there can also be *computation* latencies. It can be encoded in our model with *{begin/end}* refined operations, but one could introduce "less constant" computation latencies and pipeline stages;
- Marked graphs do not allow for control-flow alternatives and control *modes*. One reason is that, in a generalized setting such as full Petri Nets, it can no longer be asserted that token are consumed and produced at the same rate. But explicit *"branch schedules"* could maybe help regulate the branching control parts similarly to the way they control the flow rate.

Last but not least, we should soon conduct real case studies to validate the approach on industrial examples .

References

[1] F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[2] Benveniste, Caspi, Edwards, Hallbwachs, Le Guernic, and de Simone. The synchronous languages twelve years later. IEEE and INRIA/IRISA, 2003.

[3] Julien Boucaron, Jean-Vivien Millo, and Robert de Simone. Another glance at relay stations in latency-insensitive designs. In *FMGALS'05*, 2005.

[4] L.P Carloni, KL. McMillan, and A.L. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and LNCS 1633 D. Peled, editors, *Proc. of the 11th Intl. Conf. on Computer-Aided Verification (CAV)*, page 12. UC Berkeley, Cadence Design Laboratories, July 1999.

[5] Mario R. Casu and Luca Macchiarulo. A new approach to latency insensitive design. In *DAC'2004*, 2004.

[6] J. Carlier Ph. Chrétienne. *Problème d'ordonnancement: modélisation, complexité, algorithmes*. Masson, Paris, 1988.

[7] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks. In *POPL 2006 Proceedings*, January 2006.

[8] F. Commoner, Anatol W.Holt, Shimon Even, and Amir Pnueli. Marked directed graph. *Journal of Computer and System Sciences*, 5:511–523, october 1971.

[9] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, October 2004.

[10] Vincent H. Van Dongen, Guang R. Gao, and Qi Ning. A polynomial time method for optimal software pipelining. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing*, volume 634 of *LNCS*, pages p613–624, 1992.

[11] Eugene Lawler. *Combinatorial Optimization: Network and Matroids*. Holt, Rinehart and Winston, 1976.

[12] Luca P.Carloni, Kenneth L.McMillan, and Alberto L.Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[13] Luca P.Carloni, Kenneth L.McMillan, Alexander Saldanha, and Alberto L.Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *THE BEST OF ICAD*, 200x.

[14] Luca P.Carloni and Alberto L. Sangiovanni-Vincentelli. Coping with latency in soc design. *IEEE Micro*, 22(5):24–35, September/October 2002.

[15] François R.Boyer, El Mostapha Aboulhamid, Yvon Savaria, and Michel Boyer. Optimal design of synchronous circuits using software pipelining. In *Proceedings of the ICCD'98*, 1998.

Contents

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399